The FuzzyTools Component

By David Adams

Technical Note 06-19

Overview

Databases are great at finding data, provided you know exactly what you're looking for. As an example, consider the names listed below:

David Anders
Davd Anderson
David Andersen

It's easy enough for the human eye to see that these names are similar. The problem is, how do you get software to recognize and rank their similarity? Fortunately for us, large, well-funded organizations such as the US Census Bureau have spent a great deal of money and effort developing effective fuzzy matching techniques. The FuzzyTools component brings a suite of fuzzy comparisons tools to 4th Dimension. The techniques included fall into two categories: 1) phonetic (sounds-like) matching and, 2) string-distance (difference) weighting. Common uses for these tools include the following:

- Finding possible duplicates based on names or other strings that sound similar. (Phonetic matching.)
- Looking up records when the user doesn't know an exact spelling, but only a word's sound. (Phonetic matching.)
- Finding likely alternatives to a name or other string. (Distance matching.)
- Finding possible duplicates based on typos. (Distance and phonetic matching.)
- Finding possible duplicates based on names or other strings that are relatively similar. (Distance matching.)

This technical note and its sample database document and demonstrate how to use the various FuzzyTools routines, while Technical Note 06-18 **Fuzzy Matching in 4th Dimension** discusses the component's internals in more detail. Technical Note 06-20 **Data Cleaning and Deduplication** uses the component as part of an approach to improving data quality.

Note *If you want or need to rewrite or extend the existing component, Technical Note 06-18* **Fuzzy Matching in 4th Dimension** *includes the component's source code.*

Why a Component?

Components provide a way of packaging a collection of 4th Dimension resources for easy distribution. Some developers avoid building or using components because of historical problems or because the code of protected methods is not visible once installed as a component. In counterpoint to these potential drawbacks, packaging the FuzzyTools code as a component brings the following advantages:

- **Reduced Complexity**
  The source code includes over 70 methods, most of which should not be called directly. The component exposes roughly 20 methods, greatly reducing the effort of learning how to use the tool.
- **Simplified Updating**
  Using a component makes it easy to keep the FuzzyTools code up to date in multiple systems.
- **Efficient Error Testing**
  Internally, the fuzzy functions need to test that parameters, such as pointers and selector strings, are valid. Requiring each low-level method to performs these checks makes each low-level routine larger and more complex. Instead, a consolidated "gateway" routine handles parameter testing and other error checking thoroughly. If all of the inputs and preconditions are correct, then a private, low-level routine is called. Since all calls to the private routine are through a gatekeeper, the low-level routines don't need to do any testing of preconditions or parameters. This architecture reduces code bulk and complexity while preserving the benefits of rigorous error checking.

Note *The component consists entirely of project methods and doesn't add, require, link to, or alter any other resources.*

Compilation

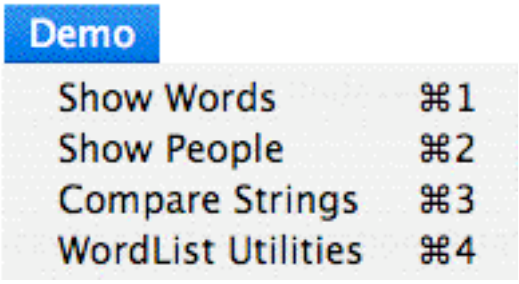
The code in the FuzzyTools component is designed to be run compiled, not interpreted. The component includes complete declarations for all variables, arrays, and parameters used. You may compile with any of 4th Dimensions compilation options, including "all variables are typed".

Internal Documentation

Each visible method in the component includes Explorer comments documenting parameters and providing a relevant code sample. Additionally, the *FuzzyTools Read Me Public* method briefly describes and documents each method.

About the Sample Database

The sample database provided includes four demonstration screens that let you experiment with all of the FuzzyTools functions and apply them to real data. We'll briefly summarize each demonstration here and document them in more detail below.



Show Words
The Show Words demonstration lets you see the results of applying all of the phonetic and distance matching routines to actual values. You can also interactively search, based on the weighting distance measures. Two sample data sets are included with the demonstration, one with about 15,000 surnames and another with about 5,000 place names. To get a better sense of how the phonetic and distance weighting tools in the component work, create a fresh data file and import some values from one of your systems into the [Sample] table. For the import, prepare a text file with two columns:

```
Word             Alpha 80
Data_Set_Name    Alpha 20
```

Show People
Roughly 500 people records with 50 duplicate pairs (100 records) are included to support a demonstration of a duplicate hunting reporting system. This topic is addressed in more detail in Technical Note 06-20 **Data Cleaning and Deduplication**.

Compare Strings
This screen lets you enter two strings and calculate their phonetic keys and similarities, using all of the functions included in the FuzzyTools component.

WordList Utilities
The sample database includes some "word list" utilities not included in the component. The word list tools are designed to compare two blocks of text based on the percentage of unique words they share in common. This percentage can be useful for comparing text or the combined values of several fields, and it is used in the duplicate hunting reporting system.

Now, let's look at the specific tools in detail. First, we'll review the phonetic matching features.

Working with Phonetic Keys

Supported Phonetic Matching Algorithms

There are a handful of well-known algorithms for generating phonetic keys for English words, including Soundex, Metaphone, Double-Metaphone, Caverphone, Phonix, and

NYIIS. Many of these algorithms have been around for decades or longer and are available in many variations. The FuzzyTools component method *Fuzzy_GetPhoneticKey* supports seven methods, summarized below:

Method	Notes
`Metaphone4`	Produces a Metaphone code of up to four characters.
`Metaphone6`	Produces a Metaphone code of up to six characters.
`Skeleton_Key`	Produces a skeleton key code of up to ten characters.
`Soundex_Knuth`	Produces a four character Soundex code using code based on Knuth's ***The Art of Computer Programming***.
`Soundex_Miracode`	Produces a four character Soundex code based on the manual system used in the 1910 US Census.
`Soundex_Simplified`	Produces a four character Soundex code based on the original system developed by the US Census in the 1880's.
`Soundex_SQLServer`	Produces a four character Soundex code emulating the behavior of the `SOUNDEX` function in MS SQLServer.

We'll discuss which algorithm to use in more detail shortly, but a good default is Metaphone4. If you need to retrieve the list of valid algorithm names within your code, use the *Fuzzy_GetPhoneticMethodTypes* function:

```
ARRAY TEXT($phoneticMethodNames_as;0)
Fuzzy_GetPhoneticMethodTypes (->$phoneticMethodNames_as)
```

Note *You may pass a pointer to a string or a text array to* Fuzzy_GetPhoneticMethodTypes*. If you pass a string array, be sure the elements are at least 18 characters long.*

Selecting a Phonetic Key Algorithm

Since FuzzyTools implements seven phonetic code generating algorithms, an obvious question is, which one to use? The answer is: "use Metaphone4 or Metaphone6". Soundex, in all its variations, is not particularly good. The Soundex code is included primarily because it is such a well-know algorithm and many existing databases include soundex encoded data. Unfortunately, Soundex codes produce a high rate of false-positives. The skeleton key algorithm is, properly speaking, not a phonetic key generating algorithm at all. It is included largely for comparison as it produces high levels of false-negatives. Metaphone4 and Metaphone6 use the same code, their only difference being the maximum length of the code produced. You may find better results with shorter or longer keys, depending on your data.

Note that all of these phonetic key generating algorithms are based on English speakers pronouncing written surnames. These algorithms are not likely to perform well for other languages or even for many variations of English. This subject is discussed in more detail in Technical Note 06-18 **Fuzzy Matching in 4th Dimension**.

Generating a Phonetic Key with *Fuzzy_GetPhoneticKey*

Fuzzy_GetPhoneticKey (Alpha 20;Alpha 80): Alpha
Fuzzy_GetPhoneticKey (Algorithm name;Base string): Phonetic code

This routine generates a phonetic ("sounds-like") code for an input string. In $1, pass the name of one of the supported phonetic key generating algorithms. The sample code below shows how to call this routine to create a phonetic key for a string stored in a field.

```
C_STRING  (4;$metaphone4_s)
$metaphone4_s:=Fuzzy_GetPhoneticKey ("Metaphone4";[Sample]Word)

If (FuzzyTools_GetLastPhoneticErrorCode=0)  ` No error
   [Sample]Metaphone4:=$metaphone4_s
End if
```

The length of the result depends on the algorithm selected, as documented below.

Algorithm	Result Length
Metaphone4	Up to 4 characters.
Metaphone6	Up to 6 characters.
Skeleton_Key	Up to 10 characters.
Soundex_Knuth	Exactly 4 characters.
Soundex_Miracode	Exactly 4 characters
Soundex_Simplified	Exactly 4 characters
Soundex_SQLServer	Exactly 4 characters

Tip *You can read the available algorithm names with* Fuzzy_GetPhoneticMethodTypes.

Reading the Algorithms with *Fuzzy_GetPhoneticMethodTypes*

Fuzzy_GetPhoneticMethodTypes (Pointer)
Fuzzy_GetPhoneticMethodTypes (->String/text array to receive names)

This routine takes a pointer to a string or text array and populates it with the names of all implemented algorithms. The *Fuzzy_GetPhoneticKey* method requires a valid name.

The example code below shows how to call this routine:

```
ARRAY  TEXT($phoneticMethodNames_at;0)
Fuzzy_GetPhoneticMethodTypes(->$phoneticMethodNames_at)
```

You may pass in a pointer to a string or text array, but, if you use a string array, be sure it is at least 18 characters long. For example:

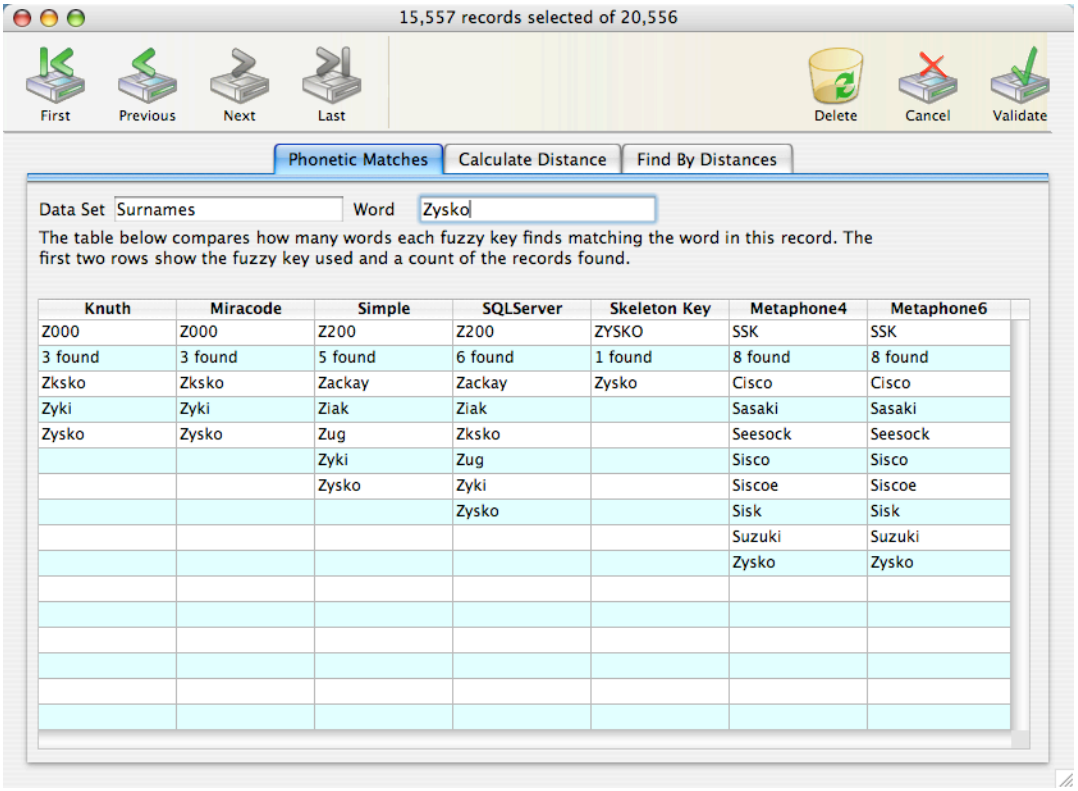
```
ARRAY  STRING(18;$phoneticMethodNames_as;0)
Fuzzy_GetDistanceMethodTypes(->$phoneticMethodNames_as)
```

If you pass in a reference to a string array with shorter elements, you will get runtime errors in compiled mode. 4th Dimension does not invoke the error handler used by FuzzyTools in response to this error.

Tip *Apart from discovering the valid values for passing to* Fuzzy_GetPhoneticKey*, the full list can be useful if, for example, you are building a GUI to test and compare the various phonetic algorithms when applied to your data.*

Phonetic Key Demonstration: Codes and Matches

To get a sense of how phonetic keys look and work, try the Show Words demonstration in the sample database. The first page of this form shows the sample word listed along with its phonetic code for each of the seven phonetic algorithms and the words in the data file that share that phonetic key, as pictured below:



Knuth	Miracode	Simple	SQLServer	Skeleton Key	Metaphone4	Metaphone6
Z000	Z000	Z200	Z200	ZYSKO	SSK	SSK
3 found	3 found	5 found	6 found	1 found	8 found	8 found
Zksko	Zksko	Zackay	Zackay	Zysko	Cisco	Cisco
Zyki	Zyki	Ziak	Ziak		Sasaki	Sasaki
Zysko	Zysko	Zug	Zksko		Seesock	Seesock
		Zyki	Zug		Sisco	Sisco
		Zysko	Zyki		Siscoe	Siscoe
			Zysko		Sisk	Sisk
					Suzuki	Suzuki
					Zysko	Zysko

Search for, or scroll through, various words to see how the different algorithms perform. Ideally, you should import some of your own data. As the example above illustrates, Metaphone often finds more reasonable matches than Soundex.

Tip *Run the database compiled for better performance.*

Phonetic Key Error Management

The routines in the FuzzyTools component install a custom error handler before performing any work. If you have another error handler installed already, it is restored at the end of any FuzzyTools routine. If an error is encountered by FuzzyTools, the method that encountered the error and an error code are set. Distinct errors and error descriptions are stored for fuzzy phonetic and fuzzy distance routines. The fuzzy phonetics error methods and errors are described below.

Fuzzy_GetLastPhoneticErrorCode
Fuzzy_GetLastPhoneticErrorCode (): Longint
Fuzzy_GetLastPhoneticErrorCode (): Error code or 0, if there was no error.

Fuzzy_GetPhoneticErrorText
Fuzzy_GetPhoneticErrorText (Longint): Text
Fuzzy_GetPhoneticErrorText (Error code): Error text

A string translation of any fuzzy phonetics error code can be read using the *Fuzzy_GetPhoneticErrorText* function. If you want the text of the last error code set, call the code shown below:

Fuzzy_GetPhoneticErrorText (*Fuzzy_GetLastPhoneticErrorCode*)

Defined Error Strings
The table below lists all defined fuzzy phonetic errors.

#	Error Text
1	Required parameter(s) not passed to Fuzzy_GetPhoneticKey.
2	Bad phonetic key method name type passed to Fuzzy_GetPhoneticKey.
3	Internal error: Phonetic key name method passed to Fuzzy_GetPhoneticKey not recognized or caught as an error.
4	Calling Fuzzy_GetPhoneticErrorText without the required error code parameter.
5	Required parameter not passed to Fuzzy_GetPhoneticMethodTypes.
6	Bad or nil pointer passed to Fuzzy_GetPhoneticMethodTypes.
7	Pointer to wrong data type passed to Fuzzy_GetPhoneticMethodTypes.

Working with Word Difference Measurements

String Difference Algorithm Styles
There are several approaches to quantifying the similarity/difference between two strings. FuzzyTools implements three different algorithm styles: weighted similarities, edit distance, and common subsequences. We'll review these briefly and then document the methods you can use to apply these features to your data.

Weighting Scoring Algorithms
There are several statistical algorithms for calculating the difference between two strings. These algorithms are commonly used to compare words in spell-checkers, strings in databases, and protein chains in DNA analysis software. The FuzzyTools *Fuzzy_GetDistancePercentage* method implements four related algorithms developed by statisticians at the US Census from 1990 and onwards, summarized below:

Method	Notes/Source
`Jaro`	The most basic algorithm.
`Winkler`	Refines the weight produced by Jaro.
`McLaughlin`	Refines the weight produced by Winkler.
`Lynch`	Refines the weight produced by McLaughlin.

These algorithms have been field-tested and refined on enormous data sets and are quite robust and powerful.

Edit Distance Algorithms

Another way to quantify the difference between two strings is by counting their differences, a measurement commonly called an "edit distance". Edit distances are a primary tool in spell-checkers and are sometimes used in database comparisons. The FuzzyTools *Fuzzy_GetEditDistanceCount* routine implements one of the best-known of these algorithms, called the "Levenshtein distance". This algorithm returns a count of how many additions, deletions and substitutions are required to transform one string into another. The more differences there are between the strings, the more steps are required to make them identical and, therefore, the higher the distance count. Identical strings require no transformations and, therefore, return a count of 0. For an example of two unlike strings, the edit distance between "kitten" and "sitting" is 3:

```
0   kitten
1   sitten    Substitute 's' for 'k'.
2   sittin    Substitute 'i' for 'e'.
3   sitting   Insert 'g' at the end of the word.
```

As this example illustrates, the two strings don't need to be of the same length to be compared. This behavior is shared by all of the FuzzyTools functions. Now let's look at how common subsequence comparisons work.

Longest Common Subsequence

Another strategy for comparing two strings, or streams of bytes, is called Longest Common Subsequence, or LCS. This technique is commonly used to compare DNA sequences, for example. This approach is somewhat similar to finding the longest matching substring. For example, imagine these two strings:

```
John Anderson
Jon Anderrsen
```

If you were finding the longest matching *substring*, you would end up with the result highlighted below:

```
John Anderson
Jon Anderrsen
```

The substring above is 5 characters long (**Ander**) out of 13 in the original strings. Converted to a percentage, that's a similarity of a bit over 38%. Just from looking at

the strings, the score seems too low. The LCS algorithm, implemented in FuzzyTools's *Fuzzy_GetLCSLength* routine, finds the longest common *subsequence* instead of the longest substring. The difference is that a subsequence ignores non-matching intervening characters. So, in comparing "Jon Anderrsen" to " John Anderson", the pattern highlighted below registers as a match (the space character matches but can't be highlighted):

John Anderson
Jon Anderr**s**e**n**

Notice that Jon and John match because the characters **J**-**o**-**n** appear, in order, within **J**-**o**-h-**n**. The **h** in John is simply ignored as a junk character. The longest matching subsequence, then, is 11 characters long (**Jon Andersn)**, giving us a similarity percentage of roughly 85%. This score agrees much more closely with how a human would rank the two strings.

Note *The two strings in the example above are the same length only to make the example easier to follow. In practice, you may compare strings of different lengths with all of the distance comparison algorithms in FuzzyTools.*

Distance Score Examples

The distances scores are easier to understand after looking at some examples. The *Fuzzy_GetDistancePercentage* routine always returns a real value between 0 and 1. Another way to look at these weights is as percentages where 1.00 = 100% agreement and 0=0% agreement. So, a score of .679 indicates a similarity of 67.9%. Below are a few sample results based on comparing words to "Adkinson" using the Lynch algorithm. I've translated the scores into percentages (score * 100) for convenience. I have also added the edit distance counts and LCS lengths for comparison.

Word	Score	%	Edit	LCS	Notes
Adkinson	1.0000	100.0	0	8	This is the base word so you should expect 100% agreement.
Adkins	0.9556	95.6	2	6	**Adkins** gets a high score because it matches the front of **Adkins**on perfectly
Atkinson	0.9488	94.8	1	7	**Atkinson** deserves a high score because it differs from the original by only one character: A**d**kinson
Atchinson	0.8882	88.2	3	6	
Apple	0.5556	55.6	7	1	
Blueberry	0.0700	7.0	9	0	If anything, this weighted score is too high.

Note *The string comparison algorithms implemented here are not biased towards English and should work well with any language. However, they may be biased towards left-to-right word order and may not prove as accurate with right-to-left word order.*

Note that the various statistical algorithms implemented in *Fuzzy_GetDistancePercentage* are not as simple as the edit distance and LCS algorithms. Internally, the statistical techniques give preferential weighting to various factors, such as similarities nearer to the front of the word. Therefore, there is a high but imperfect correlation between edit distance scores and weighted distance scores.

As an example, the Lynch algorithm considers “Adkins” more similar to “Adkinson” than to “Atkinso” while the edit distance algorithm ranks them in the opposite order. It also makes sense that differences in edit distance counts and LCS lengths are likely to be more meaningful when comparing longer strings.

Normalizing Similarity Scores

As you may have noticed in the example above, the three different algorithm types produce different scores on different scales sometimes running in opposite directions on the number line, as summarized below:

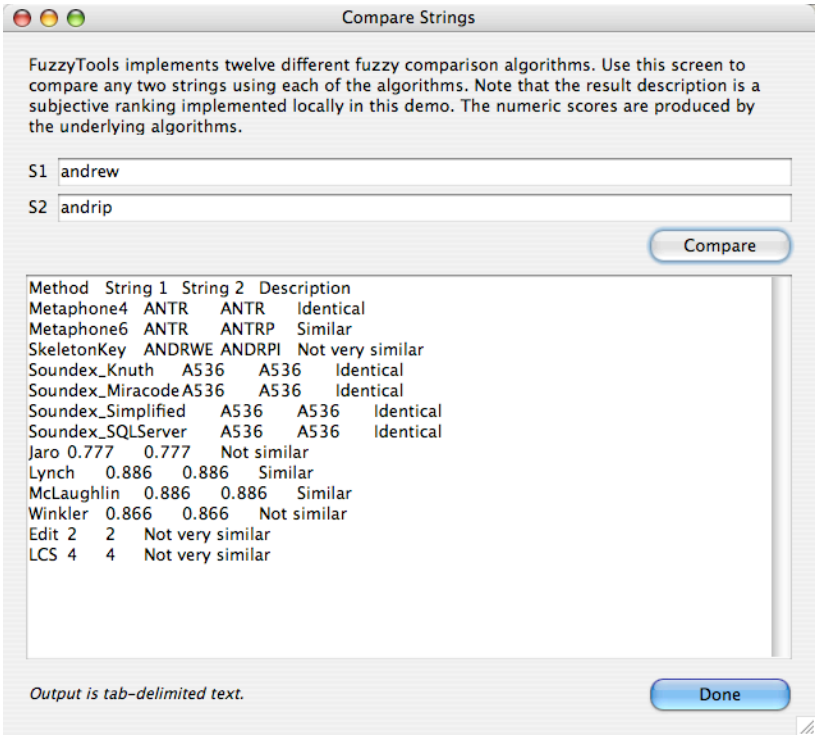
Strategy	Method	Output	Identical
Statistical Similarity	*Fuzzy_GetDistanceePercentage*	Real from 0-1	1
EditDistance	*Fuzzy_GetEditDistanceCou*	Longint count	0

	nt		
Longest Common Subsequence	*Fuzzy_GetLCSLength*	Longint length	String length

The scales used by *Fuzzy_GetEditDistanceCount* returns a longint count, *Fuzzy_GetLCSLength* returns a longint length, and *Fuzzy_GetDistancePercentage* returns a real percentage. Each of these approaches makes sense for their respective algorithms but can cause confusion when a developer is working with scores. To simplify the system, the *Fuzzy_GetDistancePercentage* routine can produce any of the six possible scores (Jaro, Winkler, McLaughlin, Lynch, Edit, and LCS) as a percentage. Internally, raw edit distance and LCS scores are converted into a percentage to make them comparable with the results from the statistical functions. This common scoring strategy makes it a lot easier to compare the different tools and to use them together. This feature is particularly handy when calling *Fuzzy_FindByDistancePercentage*, which always expects a percentage. You still have access to raw edit distance and LCS scores and the routines to convert them to percentages them, if you prefer.

Tip: Compare Strings Demonstration

It's always easier to understand how things work by using them yourself. The sample database demonstrations include many string comparison tools, including a screen that lets you enter any two strings and compare them with all of the methods, pictured below:



The Show Words demonstration's input screen also includes distance calculation and find-by-distance routines, discussed later.

Selecting a Distance Measurement Algorithm

An obvious question to ask now is, which distance measurement is best? Internally, all four weighted algorithms are closely related. Jaro is the simplest, Winkler refines Jaro, McLaughlin refines Winkler, and Lynch refines McLaughlin. Therefore, the most refined weighting algorithm is Lynch, a good bet for your default approach to weighted results. Depending on your data and how you approach setting thresholds, using the edit distance or the LCS count may be quicker and no less effective. The different algorithms produce different results and, therefore, match different related values. The example below is based on the surnames data set in the sample database. Starting with the last name “Zysko”, below are the surnames that match based on a threshold of 80%:

Method	Jaro	Lynch	McLaughlin	Winkler	Edit	LCS
Matches	2	5	3	3	2	2
	Zysko	Zysko	Zysko	Zysko	Zysko	Zysko
	Zkssko	Zkssko	Zkssko	Zkssko	Zkssko	Zkssko
		Zyki	Zyki	Zyki		
		Risko				
		Sykor				

As you can see from this sample, you can get very different results from the various algorithms. The six different algorithms locate six possible matching names, but no one algorithm finds all of them. Depending on your data, you may find one approach works far better than another or, more likely, that using a combination of approaches delivers the best results. A common strategy is to use the Metaphone phonetic matching algorithm to find likely matches and then use the edit distance or Lynch

weighting algorithms to sort the possible matches. The benefit of this approach is that Metaphone codes are short, easily indexed strings, perfect for fast searches.

The best way to find out what approach works best is to test your own data. The sample database is designed with this purpose in mind and includes several screens, discussed below, that let you experiment easily with the different tools. Below we'll look at how to use the tools in more detail.

Tip *If you have a good idea of how long your comparison strings are and how similar they should be, try calling* Fuzzy_FindByEditDistanceCount *or* Fuzzy_FindByLCSLength *directly with raw values. In some cases, this routine is significantly faster than calling* Fuzzy_FindByDistancePercentage *with a percentage.*

Generating a Distance Weight with *Fuzzy_GetDistancePercentage*

Fuzzy_GetDistancePercentage (Alpha 20; Alpha 80; Alpha 80): Real
Fuzzy_GetDistancePercentage (Algorithm name;Base string;Comparison string): Weight

This routine calculates the distance between two strings. This is a weighted measure of how closely the two strings resemble each other. A result of 1 indicates "strings considered identical" and a result of 0 indicates "strings considered entirely different". Therefore, higher scores indicate a higher degree of similarity and lower scores indicate a lower degree of similarity.

In $1, pass the name of one of the supported distance calculating algorithms:

```
Jaro
Lynch
McLaughlin
Winkler
Edit
LCS
```

The example below illustrates how to use this routine:

```
C_REAL($distance_weight)
$distance_weight:=Fuzzy_GetDistancePercentage ("Lynch";massey";"massie")
```

After calling the code shown above, $distance_weight contains the value 0.953. This score makes sense as "massey" and "massie" are very similar but not exactly the same. The edit and LCS methods are included as a convenience. Internally, edit distances is calculated by *Fuzzy_GetEditDistanceCount* and LCS is calculated by *Fuzzy_GetLCSLength*.

If you need to retrieve the list of valid algorithm names within the code, use the *Fuzzy_GetDistanceMethodTypes* function, as illustrated below:

```
ARRAY TEXT($distanceMethodNames_as;0)
Fuzzy_GetDistanceMethodTypes(->$distanceMethodNames_as)
```

Note *See also the discussions of* Fuzzy_GetEditDistanceCount *and* Fuzzy_GetLCSLength.

Generating a Distance Count with *Fuzzy_GetEditDistanceCount*

Fuzzy_GetEditDistanceCount (Alpha 80; Alpha 80): Longint
Fuzzy_GetEditDistanceCount (Base string;Comparison string): Count

This routine counts the differences between two strings and is a sum of the number of additions, deletions, and substitutions required to transform one string into another. Identical strings receive a score of 0 for "no transformations required". Two entirely unlike strings receive a score equal to the longest string. Strings with some similarity receive a score between zero and the length of the longest string. Therefore, higher scores indicated a lower degree of similarity, and lower scores signify a higher degree of similarity. The sample code below shows the routine in use:

```
C_LONGINT($distance_count)
$distance_count:=Fuzzy_GetEditDistanceCount  ("massey";"massie")
```

After calling the code shown above, $distance_count contains the value 2. This score makes sense as "massey" needs its last two letters "ey" changed to "ie" to complete the transformation into "massie".

Note that *Fuzzy_GetEditDistanceCount* scores run in the opposite direction to *Fuzzy_GetDistancePercentage*. Remember that *Fuzzy_GetDistancePercentage* can produce edit distances converted to a standard percentage of similarity. If you want to convert distance counts to percentages yourself, call *Fuzzy_EditDistanceToPercentage*.

Generating a Subsequence Length with *Fuzzy_GetLCSLength*

Fuzzy_GetLCSLength (Alpha 80; Alpha 80): Longint
Fuzzy_GetLCSLength (Base string;Comparison string): Length

This routine counts the longest shared subsequence between two strings. As discussed above, a subsequence is the longest string of matching characters found between the two strings, without regard to 'junk' characters. To repeat an earlier example, the two strings below have an LCS count of 11, the string **Jon Andersn**, including the space:

John Anderson
Jon Anderrsen

Since an LCS score is a length, the higher the number, the greater the degree of agreement between the two strings. The sample code below shows the routine in use:

```
C_LONGINT($lcs_length)
$lcs_length:=Fuzzy_GetEditDistanceCount  ("massey";"massie")
```

After calling the code shown above, $lcs_length contains the value 5. This score makes sense as the two strings have a common subsequence of **masse** (**masse**y and **mass**i**e**).

Note that *Fuzzy_GetLCSLength* returns whole integers, not percentages like *Fuzzy_GetDistancePercentage*. Remember that *Fuzzy_GetDistancePercentage* can produce LCS lengths converted to a standard percentage of similarity. If you want to convert LCS lengths to percentages yourself, call *Fuzzy_LCSLengthToPercentage*.

Reading Algorithm Names with *Fuzzy_GetDistanceMethodTypes*

Fuzzy_GetDistanceMethodTypes (Pointer)
Fuzzy_GetDistanceMethodTypes (->String/text array to receive names)

The *Fuzzy_GetDistancePercentage and Fuzzy_FindByDistancePercentage* routines requires a valid name. Use *Fuzzy_GetDistanceMethodTypes* to copy the valid selector strings to a string or text array.

The example code below shows how to call this routine:

```
ARRAY TEXT($distanceMethodNames_at;0)
Fuzzy_GetDistanceMethodTypes(->$distanceMethodNames_at)
```

You may pass in a pointer to a string or text array but, if you use a string array, be sure it is at least 10 characters long. For example:

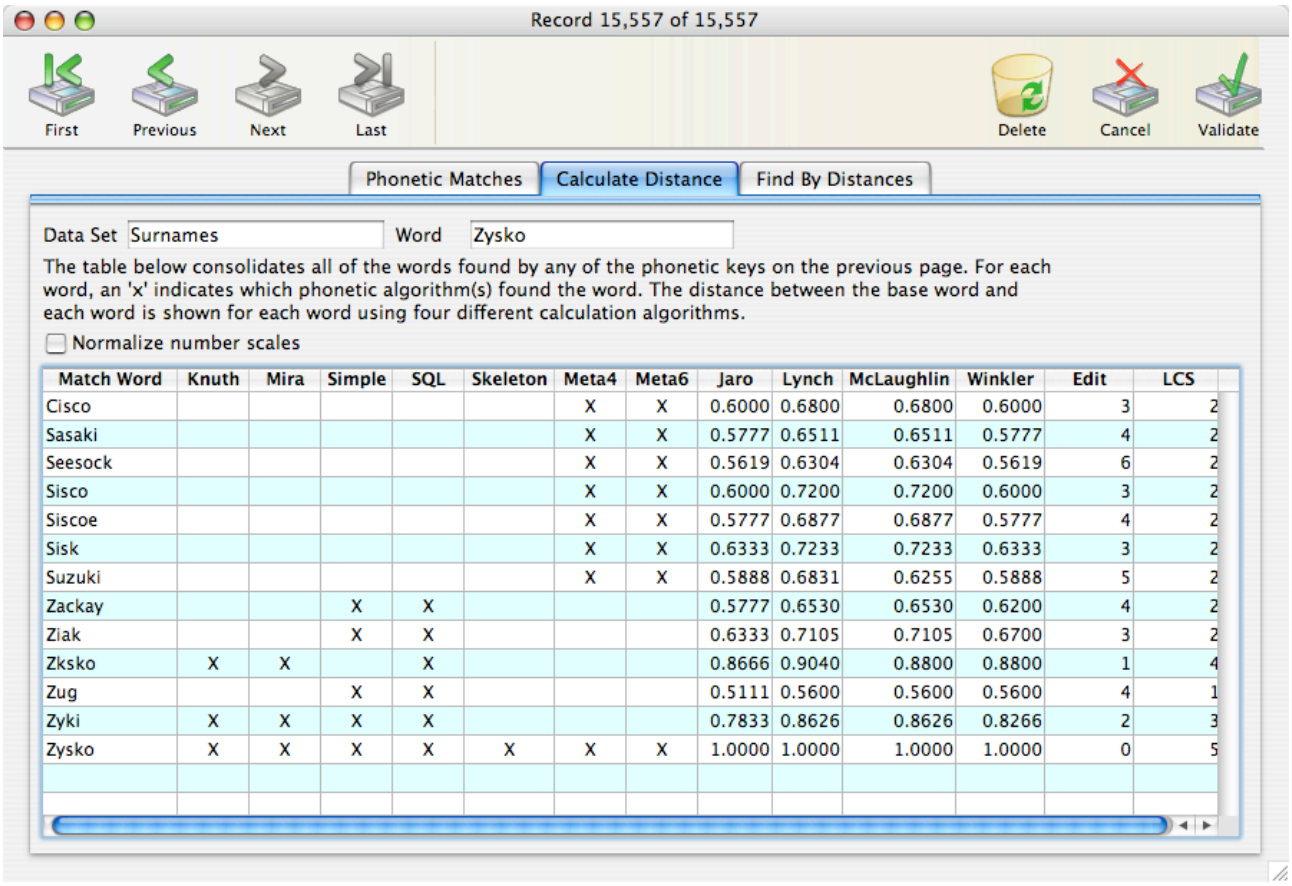
```
ARRAY STRING(10;$distanceMethodNames _as;0)
Fuzzy_GetDistanceMethodTypes (->$distanceMethodNames _as)
```

If you pass in a reference to a string array with shorter elements, you will get runtime errors in compiled mode. 4th Dimension does not invoke the error handler used by FuzzyTools in response to this error.

Tip *Apart from discovering the valid values for passing to* Fuzzy_GetDistancePercentage*, the full list can be useful if, for example, you are building GUI that includes the distance algorithm as an option.*

Distance Measurement Demonstration: Weight and Matches

To get a sense of how the distance measurements look and work, try the Show Words demonstration in the sample database. This demonstration opens a new process with a list of sample words. Open any word to view the distance results. The first page, discussed earlier, lists the seven available phonetic codes and the words that share each code. The second page consolidates the words matched by any of the seven phonetic codes and calculates their weights using each of the five distance algorithms, as illustrated below:



Match Word	Knuth	Mira	Simple	SQL	Skeleton	Meta4	Meta6	Jaro	Lynch	McLaughlin	Winkler	Edit	LCS
Cisco						X	X	0.6000	0.6800	0.6800	0.6000	3	2
Sasaki						X	X	0.5777	0.6511	0.6511	0.5777	4	2
Seesock						X	X	0.5619	0.6304	0.6304	0.5619	6	2
Sisco						X	X	0.6000	0.7200	0.7200	0.6000	3	2
Siscoe						X	X	0.5777	0.6877	0.6877	0.5777	4	2
Sisk						X	X	0.6333	0.7233	0.7233	0.6333	3	2
Suzuki						X	X	0.5888	0.6831	0.6255	0.5888	5	2
Zackay			X	X				0.5777	0.6530	0.6530	0.6200	4	2
Ziak			X	X				0.6333	0.7105	0.7105	0.6700	3	2
Zksko	X	X		X				0.8666	0.9040	0.8800	0.8800	1	4
Zug			X	X				0.5111	0.5600	0.5600	0.5600	4	1
Zyki	X	X	X	X				0.7833	0.8626	0.8626	0.8266	2	3
Zysko	X	X	X	X	X	X	X	1.0000	1.0000	1.0000	1.0000	0	5

The columns towards the left labeled Knuth, Mira, Simple, SQL, Skeleton, Meta4, and Meta6 indicate with an x if a word was matched by that phonetic algorithm. The numeric values for each distance calculation, including Jaro, Lynch, McLaughlin, Winkler, Edit, and LCS are displayed on the right. The edit distance count and LCS length values are shown as raw scores, by default. Select the **Normalize number scales** checkbox to convert these values to percentages.

Some sample data is included for your convenience but you will learn more from importing and testing your own data. As mentioned earlier, you are best off creating a fresh data file and importing a list of strings with a "data set name" into the [Sample] table:

```
Word              Alpha 80
Data_Set_Name     Alpha 20
```

Finding Data with *Fuzzy_FindByDistancePercentage*

Fuzzy_FindByDistancePercentage (Alpha 20;Alpha 80;Pointer;Real;{Boolean}) : Longint
Fuzzy_FindByDistancePercentage (Algorithm name;Base string;Search field;Threshold;{Query Selection?}) : Records found

As a convenience, the FuzzyTools component includes a routine for finding strings that are within a particular degree of similarity to a string based on a weight returned by *Fuzzy_GetDistancePercentage*. The arguments are listed below:

$1 takes a distance calculating algorithm name, as documented above in *Fuzzy_GetDistancePercentage*.

$2 takes a string (alpha 80) that serves as the basis for all comparisons.

$3 takes a pointer to the alpha field to search on. Each value is tested against the base string.

$4 takes a threshold weight. When the base string is compared with the value in the record, the values are considered a match if the similarity weight is >= the threshold weight. The higher you set the threshold, the fewer matches you will get.

Note *Remember that weighted distance scores run from 0 for entirely unlike strings upwards to 1 for identical strings. Higher scores, therefore, match more values.*

$5 optionally lets you query the current selection instead of the full table. (By default, this routine queries the entire table.) **Whenever possible, reduce the selection by some means first before calling this routine.** In order to match records, the routine needs to sequentially test the values of each record in the selection, potentially a very slow operation.

Tip *If you are unsure which algorithm to use, Lynch is a good default.*

Note that regardless of where you set the threshold, *Fuzzy_FindByDistancePercentage* still needs to compare each record in the selection or table. The overall time to perform these comparisons depends on the number of records compared, if you are running compiled, and if you are running under 4D Server. Internally, this routine uses **SELECTION TO ARRAY** to save time and reduce network traffic. If the selection is large, it is read in chunks of 255 values at a time to avoid loading huge arrays.

Finding Data with *Fuzzy_FindByEditDistanceCount*

Fuzzy_FindByEditDistanceCount (Alpha 80;Pointer;Longint;{Boolean}) : Longint
Fuzzy_FindByEditDistanceCount (Base string;Search field;Threshold;{Query Selection?}) : Records found

The FuzzyTools component includes this routine for finding strings that are within a particular distance of a base string based on a ranking from *Fuzzy_GetEditDistanceCount*. The arguments are listed below:

$1 takes a string (alpha 80) that serves as the basis for all comparisons.

$2 takes a pointer to the alpha field to search on. Each value is tested against the base string.

$3 takes a threshold count. When the base string is compared with the value in the record, the values are considered a match if the distance count is <= the threshold count. The lower you set the threshold, the fewer matches you will get.

Note *Remember that edit distance scores run from 0 for identical strings upwards for increasingly unlike strings. Higher scores, therefore, match fewer values.*

$4 optionally lets you query the current selection instead of the full table. (By default, this routine queries the entire table.) **Whenever possible, reduce the selection by some means first before calling this routine.** In order to match records, the routine needs to sequentially test the values of each record in the selection, potentially a very slow operation.

Keep in mind that regardless of what value you use as a threshold, *Fuzzy_FindByEditDistanceCount* still needs to compare each record in the current selection or table. The overall time to perform these comparisons depends on the number of records compared, if you are running compiled, and if you are running under 4D Server. Internally, this routine uses **SELECTION TO ARRAY** to save time and reduce network traffic under 4D Server. If the selection is large, it is read in chunks of 255 values at a time to avoid loading huge arrays.

Finding Data with *Fuzzy_FindByLCSLength*

Fuzzy_FindByLCSLength (Alpha 80;Pointer;Longint;{Boolean}) : Longint
Fuzzy_FindByLCSLength (Base string;Search field;Threshold;{Query Selection?}) : Records found

The FuzzyTools component includes this routine for finding strings that are within a particular distance from a base string based on a ranking from *Fuzzy_GetLCSLength*. The arguments are listed below:

$1 takes a string (alpha 80) that serves as the basis for all comparisons.

$2 takes a pointer to the alpha field to search on. Each value is tested against the base string.

$3 takes a threshold length. When the base string is compared with the value in the record, the values are considered a match if the distance count is >= the threshold length. The lower you set the threshold, the more matches you will get.

Note *Remember that LCS scores run from 0 for completely unlike strings upwards for increasingly similar strings. Higher scores, therefore, match fewer values.*

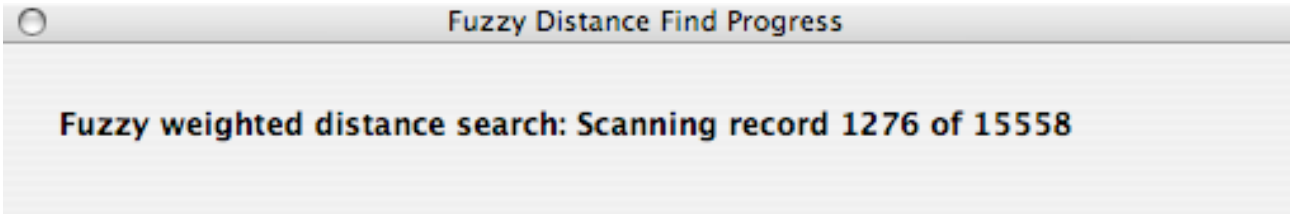
$4 optionally lets you query the current selection instead of the full table. (By default, this routine queries the entire table.) **Whenever possible, reduce the selection by some means first before calling this routine.** In order to match records, the routine needs to sequentially test the values of each record in the selection, potentially a very slow operation.

Keep in mind that regardless of what value you use as a threshold, *Fuzzy_FindByLCSLength* still needs to compare each record in the current selection or table. The overall time to perform these comparisons depends on the number of records compared, if you are running compiled, and if you are running under 4D Server. Internally, this routine uses **SELECTION TO ARRAY** to save time and reduce network traffic under 4D Server. If the selection is large, it is read in chunks of 255 values at a time to avoid loading huge arrays.

Checking Search Status with *Fuzzy_FindByDistanceGetProgress*

Fuzzy_FindByDistanceGetProgress (Longint;Pointer;Pointer)
Fuzzy_FindByDistanceGetProgress (Find process ID;Current record;Total records)

Because part of the operations performed by *Fuzzy_FindByDistancePercentage*, *Fuzzy_FindByEditDistanceCount*, and *Fuzzy_FindByLCSLength*, are sequential, they can take a long time. To speed these operation up, run compiled, run in single user, and reduce the selection. If you have to compare a great many values, it may take long enough to require a progress indicator. Instead of implementing a full progress display module, FuzzyTools provides status information on the state of a fuzzy search for you to use in your own progress system. The FuzzyTools sample database includes an example of how to integrate these values into a simple progress display, as illustrated below:



Below the parameters for *Fuzzy_FindByDistanceGetProgress* are described in more detail.

$1 is the process ID of the process running the search. Internally, this value serves as a key into the status tracking data. The system is implemented this way to support tracking the status of concurrent queries in multiple processes.

$2 takes a pointer to a numeric container for the current record being scanned. In the demonstration screen pictured above, the value is 1,276.

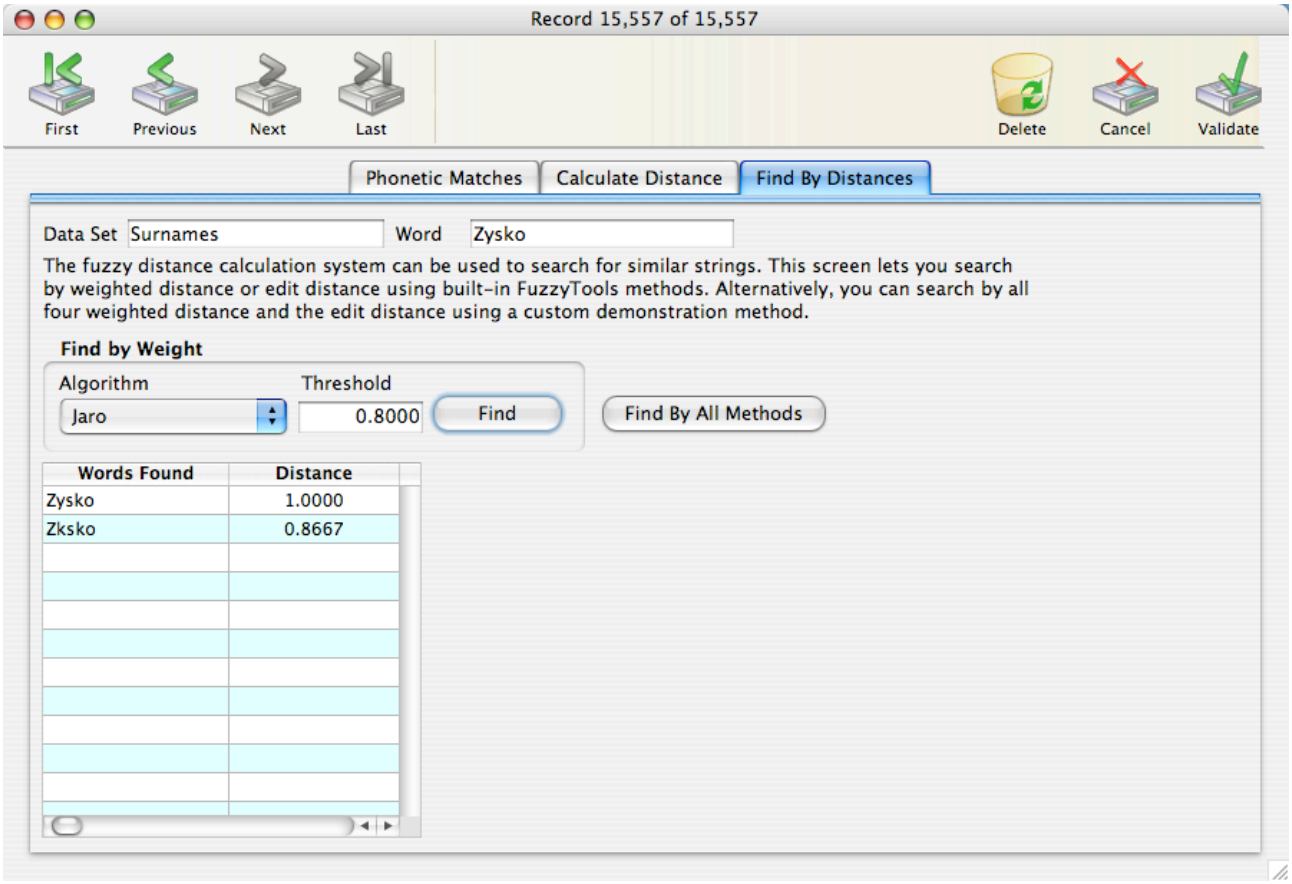
$3 takes a pointer to a numeric container for the total number of records to be scanned. In the demonstration screen pictured above, the value is 15,558.

When $2 and $3 both return -1, the search has finished.

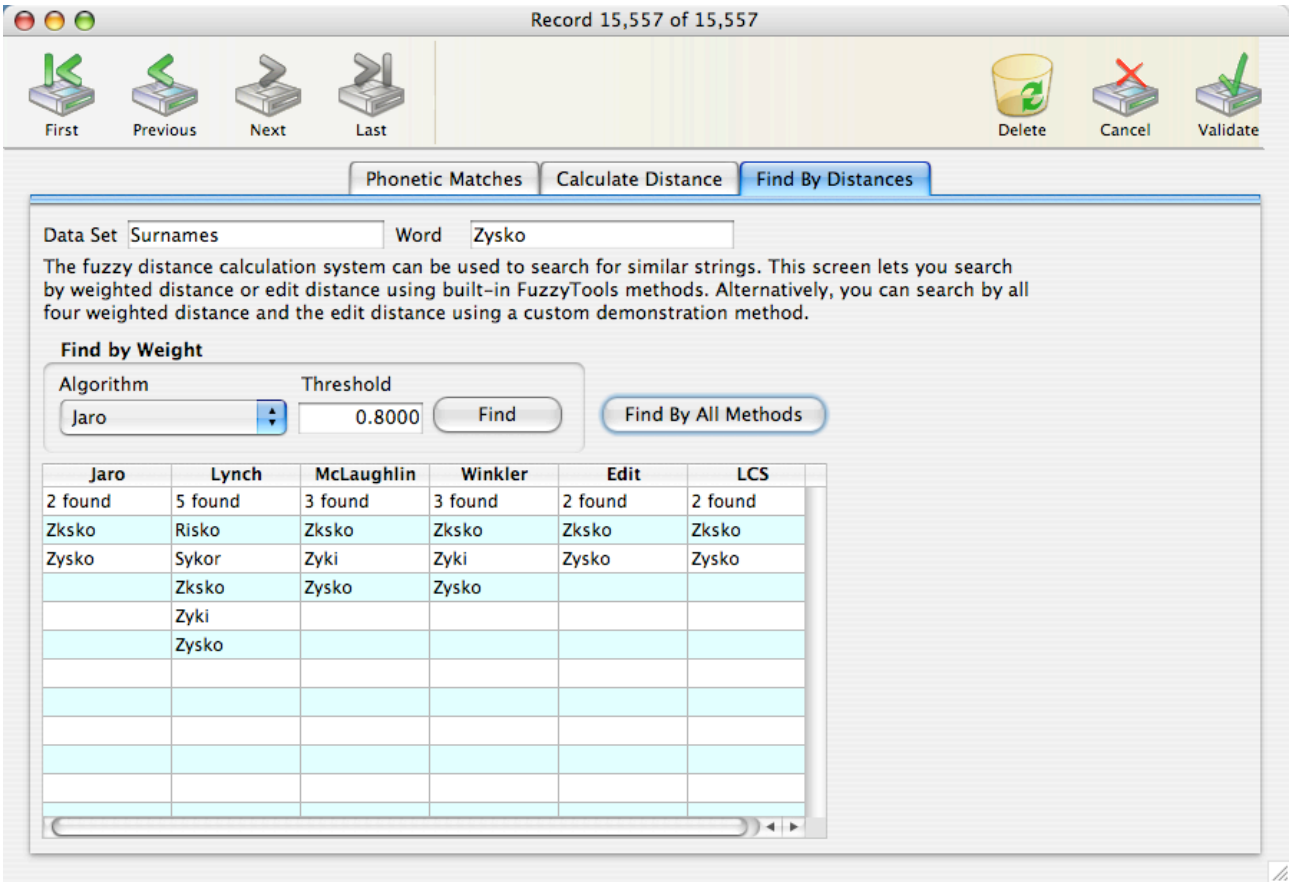
Tip *Remember that* Fuzzy_FindByDistanceGetProgress *returns two numbers, not a progress string. The exact style, formatting, and language used for the progress system is left to you.*

Find by Distance Demonstration

To get a sense of how the distance matching system works, try page three of the Show Words demonstration's input screen, pictured below. This screen lets you search on-the-fly for words that are greater than or equal to a specific degree of similarity to the current word. You can search by any of the weighting algorithms, or by all six available methods at once. (Searching by all methods is, understandably, slower.)



The entry objects on the screen let you select an algorithm and weighted threshold between 0-1.0 for *Fuzzy_FindByDistancePercentage*. The screen doesn't provide a way of calling *Fuzzy_FindByEditDistanceCount* or *Fuzzy_FindByLCSLength* directly. If you press to **Find by All Methods**, the results change to display the outcome of each of the six methods, as illustrated below:



Be forewarned that this system must perform sequential comparisons and, therefore, is slow when you have a lot of records. Try running this system in compiled mode for reasonable performance. Something you should notice is that the phonetic and distance-calculating algorithms match different records. Likewise, the various distance algorithms also match different records. The distance-calculating comparisons tend to find a great many more likely duplicates than the phonetic-code based comparisons. The advantage of the phonetic codes, however, is that they can be pre-calculated, stored, and indexed in advance.

Distance Calculation Error Management

The routines in the FuzzyTools component install a custom error handler before performing any work. If you have another error handler installed already, it is restored at the end of any FuzzyTools routine. If an error is encountered by FuzzyTools, the method that encountered the error and an error code are set. Distinct errors and error descriptions are stored for fuzzy phonetic and fuzzy distance routines. The fuzzy distance error methods and errors are described below.

Fuzzy_GetLastDistanceErrorCode

Fuzzy_GetLastDistanceErrorCode (): Longint
Fuzzy_GetLastDistanceErrorCode (): Error code or 0, if there was no error.

Fuzzy_GetDistanceErrorText

Fuzzy_GetDistanceErrorText (Longint): Text
Fuzzy_GetDistanceErrorText (Error code): Error text

A string translation of any fuzzy phonetics error code can be read using the *Fuzzy_GetDistanceErrorText* function. If you want the text of the last error code set, call the code shown below:

Fuzzy_GetDistanceErrorText (*Fuzzy_GetLastDistanceErrorCode*)

Defined Error Strings

The table below lists all defined fuzzy distance errors.

#	Error Text
1	Required parameter(s) not passed to Fuzzy_GetDistancePercentage.
2	Bad distance calculation method type passed to Fuzzy_GetDistancePercentage.
3	Internal error: Distance calculation method passed to Fuzzy_GetDistancePercentage not recognized or caught as an error.
4	Calling Fuzzy_GetDistanceErrorText without the required error code parameter.
5	Required parameter not passed to Fuzzy_GetDistanceMethodTypes.
6	Bad or nil pointer passed to Fuzzy_GetDistanceMethodTypes.
7	Pointer to wrong data type passed to Fuzzy_GetDistanceMethodTypes.
8	Required parameter(s) not passed to Fuzzy_FindByDistancePercentage.
9	Bad distance calculation method type passed to Fuzzy_FindByDistancePercentage.
10	Bad or nil pointer passed to Fuzzy_FindByDistancePercentage.
11	Pointer to wrong data type passed to Fuzzy_FindByDistancePercentage. String field pointer expected.
12	Empty base word passed to Fuzzy_FindByDistancePercentage.
13	Out of range threshold weight passed to Fuzzy_FindByDistancePercentage. A real between 0 and 1 expected.
14	Internal error: Distance calculation method passed to Fuzzy_FindByDistancePercentage not recognized or caught as an error.
15	Required parameter not passed to Fuzzy_FindByDistanceGetProgress.
16	Bad or nil pointer passed to Fuzzy_FindByDistanceGetProgress.
17	Pointer to wrong data type passed to Fuzzy_FindByDistanceGetProgress.
18	Required parameter(s) not passed to Fuzzy_FindByEditDistanceCount.
19	Bad or nil pointer passed to Fuzzy_FindByEditDistanceCount.
20	Pointer to wrong data type passed to Fuzzy_FindByEditDistanceCount. String field pointer expected.
21	Empty base word passed to Fuzzy_FindByEditDistanceCount.
22	Out of range threshold distance passed to Fuzzy_FindByEditDistanceCount. A longint of 0 or higher expected.
23	Required parameter(s) not passed to Fuzzy_GetEditDistanceCount.
24	Required parameter(s) not passed to Fuzzy_GetLCSLength.
25	Required parameter(s) not passed to Fuzzy_FindByLCSLength.
26	Bad or nil pointer passed to Fuzzy_FindByLCSLength.
27	Pointer to wrong data type passed to Fuzzy_FindByLCSLength. String field pointer expected.
28	Empty base word passed to Fuzzy_FindByLCSLength.
29	Out of range threshold distance passed to Fuzzy_FindByLCSLength. A longint of 0 or higher expected.
30	Required parameter(s) not passed to Fuzzy_GetEditDistanceCountAsPercent.
31	Bad value passed to Fuzzy_GetEditDistanceCountAsPercent. Non-negative integer expected.
32	Required parameter(s) not passed to Fuzzy_GetLCSAsPercent.
33	Bad value passed to Fuzzy_GetLCSAsPercent. Non-negative integer expected.

Fuzzy Matching Duplicate Records

Finding duplicate records is the primary reason to use fuzzy matching in a typical database. This subject is examined in Technical Note 06-20 **Data Cleaning and Deduplication**. If you want to get an idea of what kind of results fuzzy matching can deliver, open the Show People demonstration in the sample database. The [Person] table includes 500 records of which 50 pairs (100 records) are duplicates. Press the "Report Duplicates" button to start a duplicate comparison process. When the duplicate checking code finishes, a results window appears with a textual summary of the comparisons performed and any duplicates found. A sample of the report's output, formatted for clarity, is shown below:

```
------------------------------------------------------------
Possible Duplicates Report
------------------------------------------------------------

Seconds taken:                      321
Records tested:                     500
Record comparisons:                 124,750
Possible duplicate pairs:           53

Values are sorted by similarity weight.

A Note About WordList Scores:
When a possible duplicate is found, a WordList comparison percentage is calculated and
included in the results for reference. In this example, the score is derived from the
percentage of common words in the combined values of the First_Name, Last_Name, Street,
and ZIP_Code fields.

Possible duplicate record pair #1
Overall similarity score:           165
WordList similarity score:          100.00%

ID:                 246                             218
First:              Sawyer                          Sawyer
Last:               McDonald                        McDonald
Street:             201 Lake Rd.                    201 Lake Rd.
Zip:                19859                           19859
Phone:              369-601-7741                    369-601-7741
Email:              Sawyer.McDonald@hotmail.com     Sawyer.McDonald@hotmail.cOm
```

Tip *Remember that comparing 500 records can take some time, so run the demonstration compiled.*

Summary

The FuzzyTools component implements a variety of phonetic and string-difference measurement algorithms to assist developers needing phonetic lookups or fuzzy comparisons for data matching. The internals of the component are explained in more detail in Technical Note 06-18 **Fuzzy Matching in 4th Dimension** and ideas for how to use fuzzy matching to improve data quality are discussed in Technical Note 06-20 **Data Cleaning and Deduplication**. A sample database is provided that illustrates the component's features and provides a test-bed for experimenting with data from your own systems.